INTERNATIONAL STANDARD

ISO/IEC 16262

First edition
1998-12-15

Information technology — ECMAScript language specification

Technologies de l'information — Spécifications du langage ECMAScript

iTeh STANDARD PREVIEW
(standards.iteh.ai)

ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

Reference number
ISO/IEC 16262:1998(E)

Contents

1 Scope 1

2 Conformance 1

3 Normative References 1

4 Overview 1
4.1 Web Scripting 2
4.2 Language Overview 2
4.2.1 Objects 2
4.3 Definitions 3
4.3.1 Type 3
4.3.2 Primitive value 3
4.3.3 Object 4
4.3.4 Constructor 4
4.3.5 Prototype 4
4.3.6 Native object 4
4.3.7 Built-in object 4
4.3.8 Host object 4
4.3.9 Undefined value 4
4.3.10 Undefined type 4
4.3.11 Null value 4
4.3.12 Null type 4
4.3.13 Boolean value 4
4.3.14 Boolean type 4
4.3.15 Boolean object 4
4.3.16 String value 5
4.3.17 String type 5
4.3.18 String object 5
4.3.19 Number value 5
4.3.20 Number type 5
4.3.21 Number object 5
4.3.22 Infinity 5
4.3.23 NaN 5

5 Notational Conventions 5
5.1 Syntactic and Lexical Grammars 5
5.1.1 Context-Free Grammars 5
5.1.2 The lexical grammar 5
5.1.3 The numeric string grammar 6
5.1.4 The syntactic grammar 6

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

© ISO/IEC 1998

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

5.1.5 Grammar Notation 6
5.2 Algorithm Conventions 8

6 Source Text 9

7 Lexical Conventions 9
7.1 White Space 9
7.2 Line Terminators 10
7.3 Comments 10
7.4 Tokens 11
7.4.1 Reserved Words 11
7.4.2 Keywords 11
7.4.3 Future Reserved Words 12
7.5 Identifiers 12
7.6 Punctuators 13
7.7 Literals 13
7.7.1 Null Literals 13
7.7.2 Boolean Literals 13
7.7.3 Numeric Literals 14
7.7.4 String Literals 16
7.8 Automatic semicolon insertion 19
7.8.1 Rules of automatic semicolon insertion 19
7.8.2 Examples of Automatic Semicolon Insertion 20

8 Types 21
8.1 The Undefined type 21
8.2 The Null type 21
8.3 The Boolean type 21
8.4 The String type 21
8.5 The Number type 21
8.6 The Object type 23
8.6.1 Property attributes 23
8.6.2 Internal Properties and Methods 23
8.7 The Reference Type 25
8.7.1 GetBase(V) 26
8.7.2 GetPropertyName(V) 26
8.7.3 GetValue(V) 26
8.7.4 PutValue(V, W) 26
8.8 The List type 26
8.9 The Completion Type 26

9 Type Conversion 27
9.1 ToPrimitive 27
9.2 ToBoolean 27
9.3 ToNumber 28
9.3.1 ToNumber Applied to the String Type 28
9.4 ToInteger 31
9.5 ToInt32: (signed 32 bit integer) 31
9.6 ToUint32: (unsigned 32 bit integer) 31
9.7 ToUint16: (unsigned 16 bit integer) 32
9.8 ToString 32
9.8.1 ToString Applied to the Number Type 32
9.9 ToObject 33

10 Execution Contexts 33
10.1 Definitions 33
10.1.1 Function Objects 33
10.1.2 Types of Executable Code 34

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

10.1.3 Variable instantiation 34
10.1.4 Scope Chain and Identifier Resolution 35
10.1.5 Global Object 35
10.1.6 Activation object 35
10.1.7 This 35
10.1.8 Arguments Object 36
10.2 Entering An Execution Context 36
10.2.1 Global Code 36
10.2.2 Eval Code 36
10.2.3 Function and Anonymous Code 36
10.2.4 Implementation-supplied Code 37

11 Expressions 37
11.1 Primary Expressions 37
11.1.1 The this **keyword** 37
11.1.2 Identifier reference 37
11.1.3 Literal reference 37
11.1.4 The Grouping Operator 37
11.2 Left-Hand-Side Expressions 38
11.2.1 Property Accessors 38
11.2.2 The new **operator** 39
11.2.3 Function Calls 39
11.2.4 Argument Lists 39
11.3 Postfix expressions 40
11.3.1 Postfix increment operator 40
11.3.2 Postfix decrement operator 40
11.4 Unary operators 41
11.4.1 The delete **operator** 41
11.4.2 The void **operator** 41
11.4.3 The typeof **operator** 41
11.4.4 Prefix increment operator 42
11.4.5 Prefix decrement operator 42
11.4.6 Unary + operator 42
11.4.7 Unary - operator 42
11.4.8 The bitwise NOT operator (~) 42
11.4.9 Logical NOT operator (!) 43
11.5 Multiplicative operators 43
11.5.1 Applying the * operator 43
11.5.2 Applying the / operator 43
11.5.3 Applying the % operator 44
11.6 Additive operators 45
11.6.1 The addition operator (+) 45
11.6.2 The subtraction operator (-) 45
11.6.3 Applying the additive operators (+, -) to numbers 45
11.7 Bitwise shift operators 46
11.7.1 The left shift operator (<<) 46
11.7.2 The signed right shift operator (>>) 46
11.7.3 The unsigned right shift operator (>>>) 47
11.8 Relational operators 47
11.8.1 The less-than operator (<) 47
11.8.2 The greater-than operator (>) 47
11.8.3 The less-than-or-equal operator (<=) 47
11.8.4 The greater-than-or-equal operator (>=) 48
11.8.5 The abstract relational comparison algorithm 48
11.9 Equality operators 48
11.9.1 The equals operator (==) 49
11.9.2 The does-not-equals operator (!=) 49

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

11.9.3 The abstract equality comparison algorithm 49
11.10 Binary bitwise operators 50
11.11 Binary logical operators 50
11.12 Conditional operator (?:) 51
11.13 Assignment operators 51
11.13.1 Simple Assignment (=) 52
11.13.2 Compound assignment (op= **)** 52
11.14 Comma operator (,) 52

12 Statements 52
12.1 Block 53
12.2 Variable statement 53
12.3 Empty statement 54
12.4 Expression statement 54
12.5 The if **statement** 55
12.6 Iteration statements 55
12.6.1 The while **statement** 55
12.6.2 The for **statement** 56
12.6.3 The for..in **statement** 57
12.7 The continue **statement** 57
12.8 The break **statement** 58
12.9 The return **statement** 58
12.10 The with **statement** 58

13 Function Definition 59

14 Program 59

15 Native ECMAScript objects 60
15.1 The Global Object 61
15.1.1 Value properties of the Global Object 61
15.1.2 Function properties of the Global Object 61
15.1.3 Constructor Properties of the Global Object 64
15.1.4 Other Properties of the Global Object 64
15.2 Object Objects 64
15.2.1 The Object Constructor Called as a Function 64
15.2.2 The Object Constructor 64
15.2.3 Properties of the Object Constructor 65
15.2.4 Properties of the Object Prototype Object 65
15.2.5 Properties of Object Instances 65
15.3 Function Objects 65
15.3.1 The Function Constructor Called as a Function 65
15.3.2 The Function Constructor 65
15.3.3 Properties of the Function Constructor 67
15.3.4 Properties of the Function Prototype Object 67
15.3.5 Properties of Function Instances 67
15.4 Array Objects 67
15.4.1 The Array Constructor Called as a Function 67
15.4.2 The Array Constructor 68
15.4.3 Properties of the Array Constructor 68
15.4.4 Properties of the Array Prototype Object 68
15.4.5 Properties of Array Instances 71
15.5 String Objects 71
15.5.1 The String Constructor Called as a Function 71
15.5.2 The String Constructor 71
15.5.3 Properties of the String Constructor 72
15.5.4 Properties of the String Prototype Object 72
15.5.5 Properties of String Instances 75

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

15.6 Boolean Objects **76**
15.6.1 The Boolean Constructor Called as a Function **76**
15.6.2 The Boolean Constructor **76**
15.6.3 Properties of the Boolean Constructor **76**
15.6.4 Properties of the Boolean Prototype Object **76**
15.6.5 Properties of Boolean Instances **77**
15.7 Number Objects **77**
15.7.1 The Number Constructor Called as a Function **77**
15.7.2 The Number Constructor **77**
15.7.3 Properties of the Number Constructor **77**
15.7.4 Properties of the Number Prototype Object **78**
15.7.5 Properties of Number Instances **78**
15.8 The Math Object **78**
15.8.1 Value Properties of the Math Object **78**
15.8.2 Function Properties of the Math Object **79**
15.9 Date Objects **84**
15.9.1 Overview of Date Objects and Definitions of Internal Operators **84**
15.9.2 The Date Constructor Called As a Function **87**
15.9.3 The Date Constructor **88**
15.9.4 Properties of the Date Constructor **90**
15.9.5 Properties of the Date Prototype Object **93**
15.9.6 Properties of Date Instances **98**
16 Errors **98**

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 16262 was prepared by ECMA (as ECMA-262) and was adopted, under a special “fast-track procedure”, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

iTeh STANDARD PREVIEW
(standards.iteh.ai)

ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

Information technology - ECMAScript language specification

1 Scope

This Standard defines the ECMAScript scripting language.

2 Conformance

A conforming implementation of ECMAScript must provide and support all the types, values, objects, properties, functions, and program syntax described in this specification.

A conforming implementation of this International Standard shall interpret characters in conformance with the Unicode Standard, Version 2.0, and ISO/IEC 10646-1 with UCS-2 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300.

A conforming implementation of ECMAScript is permitted to provide additional types, values, objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of ECMAScript is permitted to provide properties not described in this specification, and values for those properties, for objects that are described in this specification.

A conforming implementation of ECMAScript is permitted to support program syntax not described in this specification. In particular, a conforming implementation of ECMAScript is permitted to support program syntax that makes use of the “future reserved words” listed in 7.4.3.

3 Normative References

iTeh STANDARD PREVIEW
(standards.iteh.ai)

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

ISO/IEC 9899:1990, *Programming languages — C.*

ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.*

ISO/IEC 646:1991, *Information technology – ISO 7-bit coded character set for information interchange.*

Unicode Inc. (1996), The Unicode Standard™, Version 2.0. ISBN: 0-201-48345-9, Addison-Wesley Publishing Co., Menlo Park, California.

ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, New York (1985).

4 Overview

This clause contains a non-normative overview of the ECMAScript language.

ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment. ECMAScript as defined here is not intended to be computationally self-sufficient; indeed, there are no provisions in this specification for input of external data or output of computed results. Instead, it is expected that the computational environment of an ECMAScript program will provide not only the objects and other facilities described in this specification but also certain environment-specific *host* objects, whose description and behaviour are beyond the scope of this specification except to indicate that they may provide certain properties that can be accessed and certain functions that can be called from an ECMAScript program.

A ***scripting language*** is a programming language that is used to manipulate, customise, and automate the facilities of an existing system. In such systems, useful functionality is already available through a user interface, and the scripting language is a mechanism for exposing that functionality to program control. In this way, the existing system is said to provide a host environment of objects and facilities, which completes the capabilities of the scripting language. A scripting language is intended for use by both professional and non-professional programmers, and therefore there may be a number of informalities built into the language.

ECMAScript was originally designed to be a ***Web scripting language***, providing a mechanism to enliven Web pages in browsers and to perform server computation as part of a Web-based client-server architecture. ECMAScript can provide core scripting capabilities for a variety of host environments, and therefore the core scripting language is specified in this document apart from any particular host environment.

Some of the facilities of ECMAScript are similar to those used in other programming languages; in particular Java™ and Self, as described in:

Gosling, James, Bill Joy and Guy Steele. The Java Language Specification. Addison Wesley Publishing Co., 1996.

Ungar, David, and Smith, Randall B. Self: The Power of Simplicity. OOPSLA '87 Conference Proceedings, pp. 227–241, Orlando, FL, October, 1987.

4.1 Web Scripting

A web browser provides an ECMAScript host environment for client-side computation including, for instance, objects that represent windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies, and input/output. Further, the host environment provides a means to attach scripting code to events such as change of focus, page and image loading, unloading, error and abort, selection, form submission, and mouse actions. Scripting code appears within the HTML and the displayed page is a combination of user interface elements and fixed and computed text and images. The scripting code is reactive to user interaction and there is no need for a main program.

A web server provides a different host environment for server-side computation including objects representing requests, clients, and files; and mechanisms to lock and share data. By using browser-side and server side scripting together it is possible to distribute computation between the client and server while providing a customised user interface for a Web-based application.

Each Web browser and server that supports ECMAScript supplies its own host environment, completing the ECMAScript execution environment.

iTeh STANDARD PREVIEW
(standards.iteh.ai)

4.2 Language Overview

The following is an informal overview of ECMAScript—not all parts of the language are described. This overview is not part of the standard proper.

ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a9/iso-iec-16262-1998

ECMAScript is object-based: basic language and host facilities are provided by objects, and an ECMAScript program is a cluster of communicating objects. An ECMAScript ***object*** is an unordered collection of ***properties*** each with 0 or more ***attributes*** which determine how each property can be used—for example, when the ReadOnly attribute for a property is set to true, any attempt by executed ECMAScript code to change the value of the property has no effect. Properties are containers that hold other objects, ***primitive values***, or ***methods***. A primitive value is a member of one of the following built-in types: **Undefined**, **Null**, **Boolean**, **Number**, and **String**; an object is a member of the remaining built-in type **Object**; and a method is a function associated with an object via a property.

ECMAScript defines a collection of ***built-in objects*** which round out the definition of ECMAScript entities. These built-in objects include the **Global** object, the **Object** object, the **Function** object, the **Array** object, the **String** object, the **Boolean** object, the **Number** object, the **Math** object, and the **Date** object.

ECMAScript also defines a set of built-in ***operators*** that may not be, strictly speaking, functions or methods. ECMAScript operators include various unary operations, multiplicative operators, additive operators, bitwise shift operators, relational operators, equality operators, binary bitwise operators, binary logical operators, assignment operators, and the comma operator.

ECMAScript syntax intentionally resembles Java syntax. ECMAScript syntax is relaxed to enable it to serve as an easy-to-use scripting language. For example, a variable is not required to have its type declared nor are types associated with properties, and defined functions are not required to have their declarations appear textually before calls to them.

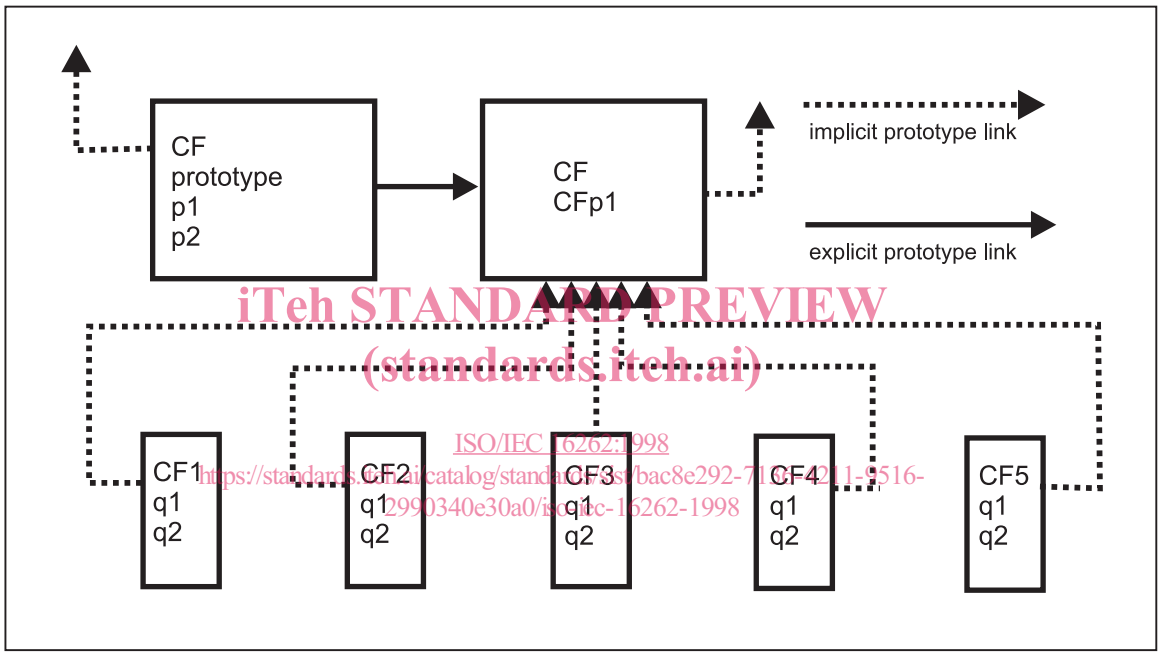
4.2.1 Objects

ECMAScript does not contain proper classes such as those in C++, Smalltalk, or Java, but rather, supports ***constructors*** which create objects by executing code that allocates storage for the objects and initialises all or part of them by assigning initial values to their properties. All functions including constructors are objects, but not all objects are constructors. Each constructor has a **Prototype** property that is used to implement ***prototype-based inheritance*** and ***shared properties***. Objects are created by using constructors in **new** expressions; for example, `new String("A String")` creates a new string object. Invoking a constructor without using **new** has consequences that depend on the constructor. For example, `String("A String")` produces a primitive string, not an object.

ECMAScript supports *prototype-based inheritance*. Every constructor has an associated prototype, and every object created by that constructor has an implicit reference to the prototype (called the *object•s prototype*) associated with its constructor. Furthermore, a prototype may have a non-null implicit reference to its prototype, and so on; this is called the *prototype chain*. When a reference is made to a property in an object, that reference is to the property of that name in the first object in the prototype chain that contains a property of that name. In other words, first the object mentioned directly is examined for such a property; if that object contains the named property, that is the property to which the reference refers; if that object does not contain the named property, the prototype for that object is examined next; and so on.

In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behaviour. In ECMAScript, the state and methods are carried by objects, and structure, behaviour, and state are all inherited.

All objects that do not directly contain a particular property that their prototype contains share that property and its value. The following diagram illustrates this:



iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7[illegible]4211-9516-12990340e30a0/iso-iec-16262-1998

CF is a constructor (and also an object). Five objects have been created by using new expressions: CF1, CF2, CF3, CF4, and CF5. Each of these objects contains properties named q1 and q2. The dashed lines represent the implicit prototype relationship; so, for example, CF3's prototype is CFp. The constructor, CF, has two properties itself, named p1 and p2, which are not visible to CFp, CF1, CF2, CF3, CF4, or CF5. The property named CFp1 in CFp is shared by CF1, CF2, CF3, CF4, and CF5, as are any properties found in CFp's implicit prototype chain which are not named q1, q2, or CFp1. Notice that there is no implicit prototype link between CFp and CF.

Unlike class-based object languages, properties can be added to objects dynamically by assigning values to them. That is, constructors are not required to name or assign values to all or any of the constructed object's properties. In the above diagram, one could add a new shared property for CF1, CF2, CF3, CF4, and CF5 by assigning a new value to the property in CFp.

4.3 Definitions

The following are informal definitions of key terms associated with ECMAScript.

4.3.1 Type

A ***type*** is a set of data values.

4.3.2 Primitive value

A ***primitive value*** is a member of one of the types **Undefined**, **Null**, **Boolean**, **Number**, or **String**. A primitive value is a datum that is represented directly at the lowest level of the language implementation.

4.3.3 Object

An ***object*** is a member of the type **Object**. It is an unordered collection of properties each of which contains a primitive value, object, or function. A function stored in a property of an object is called a method.

4.3.4 Constructor

A ***constructor*** is a function object that creates and initialises objects. Each constructor has an associated prototype object that is used to implement inheritance and shared properties.

4.3.5 Prototype

A ***prototype*** is an object used to implement structure, state, and behaviour inheritance in ECMAScript. When a constructor creates an object, that object implicitly references the constructor•s associated prototype for the purpose of resolving property references. The constructor's associated prototype can be referenced by the program expression `constructor.prototype`, and properties added to an object's prototype are shared, through inheritance, by all objects sharing the prototype.

4.3.6 Native object

A ***native object*** is any object supplied by an ECMAScript implementation independent of the host environment. Standard native objects are defined in this specification. Some native objects are built-in; others may be constructed during the course of execution of an ECMAScript program.

4.3.7 Built-in object

A ***built-in object*** is any object supplied by an ECMAScript implementation, independent of the host environment, which is present at the start of the execution of an ECMAScript program. Standard built-in objects are defined in this specification, and the ECMAScript implementation may specify and define others. Every built-in object is a native object.

iTeh STANDARD PREVIEW
(standards.iteh.ai)

4.3.8 Host object

A ***host object*** is any object supplied by the host environment to complete the execution environment of ECMAScript. Any object that is not native is a host object.

ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

4.3.9 Undefined value

The ***undefined value*** is a primitive value used when a variable has not been assigned a value.

4.3.10 Undefined type

The type **Undefined** has exactly one value, called **undefined**.

4.3.11 Null value

The ***null value*** is a primitive value that represents the null, empty, or non-existent reference.

4.3.12 Null type

The type **Null** has exactly one value, called **null**.

4.3.13 Boolean value

A ***boolean value*** is a member of the type **Boolean** and is one of two unique values, **true** and **false**.

4.3.14 Boolean type

The type **Boolean** represents a logical entity and consists of exactly two unique values. One is called **true** and the other is called **false**.

4.3.15 Boolean object

A ***boolean object*** is a member of the type **Object** and is an instance of the built-in Boolean object. That is, a boolean object is created by using the Boolean constructor in a new expression, supplying a boolean as an argument. The resulting object has an implicit (unnamed) property that is the boolean. A boolean object can be coerced to a boolean value. A boolean object can be used anywhere a boolean value is expected.

This is an example of one of the conveniences built into ECMAScript—in this case, the purpose is to accommodate programmers of varying backgrounds. Those familiar with imperative or procedural programming languages may find boolean, string and number values more natural, while those familiar with object-oriented languages may find boolean, string and number objects more intuitive.

4.3.16 String value

A ***string value*** is a member of the type **String** and is a finite ordered sequence of zero or more Unicode characters.

4.3.17 String type

The type **String** is the set of all finite ordered sequences of zero or more Unicode characters.

4.3.18 String object

A ***string object*** is a member of the type **Object** and is an instance of the built-in String object. That is, a string object is created by using the String constructor in a new expression, supplying a string as an argument. The resulting object has an implicit (unnamed) property that is the string. A string object can be coerced to a string value. A string object can be used anywhere a string value is expected.

4.3.19 Number value

A ***number value*** is a member of the type **Number** and is a direct representation of a number.

4.3.20 Number type

The type **Number** is a set of values representing numbers. In ECMAScript the set of values represent the double-precision 64-bit format IEE 754 values including the special “Not-a-Number” (NaN) values, positive infinity, and negative infinity.

4.3.21 Number object

A ***number object*** is a member of the type **Object** and is an instance of the built-in Number object. That is, a number object is created by using the Number constructor in a new expression, supplying a number as an argument. The resulting object has an implicit (unnamed) property that is the number. A number object can be coerced to a number value. A number object can be used anywhere a number value is expected. Note that a number object can have shared properties by adding them to the Number prototype.

iTeh STANDARD PREVIEW
(standards.iteh.ai)

4.3.22 Infinity

The primitive value **Infinity** represents the positive infinite number value.

ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2000340e30a0/iso-iec-16262-1998

4.3.23 NaN

The primitive value **NaN** represents the set of IEEE Standard “Not-a-Number” values.

5 Notational Conventions

5.1 Syntactic and Lexical Grammars

This clause describes the context-free grammars used in this specification to define the lexical and syntactic structure of an ECMAScript program.

5.1.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified alphabet.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the (perhaps infinite) set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

5.1.2 The lexical grammar

A *lexical grammar* for ECMAScript is given in clause 7. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *InputElement*, that describe how sequences of Unicode characters are translated into a sequence of input elements.

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for ECMAScript and are called ECMAScript *tokens*. These tokens are the reserved words, identifiers, literals, and punctuators of the ECMAScript language. Moreover, line terminators, although not considered to be tokens, also become part of the stream of input elements and guide the process of automatic semicolon insertion (see 7.8). Simple white space and single-line comments are simply discarded and do not appear in the stream of input elements for the syntactic grammar. A *MultiLineComment* (that

is, a comment of the form "`/*…*/`" regardless of whether it spans more than one line) is likewise simply discarded if it contains no line terminator; but if a multi-line comment contains one or more line terminators, then it is replaced by a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.

Productions of the lexical grammar are distinguished by having two colons "`::`" as separating punctuation.

5.1.3 The numeric string grammar

A second grammar is used for translating strings into numeric values. This grammar is similar to the part of the lexical grammar having to do with numeric literals and has as its terminal symbols the characters of the Unicode character set. This grammar appears in clause 9.3.1.

Productions of the numeric string grammar are distinguished by having three colons "`:::`" as punctuation.

5.1.4 The syntactic grammar

The *syntactic grammar* for ECMAScript is given in clauses 11, 12, 13, and 14. This grammar has ECMAScript tokens defined by the lexical grammar as its terminal symbols (see 5.1.2). It defines a set of productions, starting from the goal symbol *Program*, that describe how sequences of tokens can form syntactically correct ECMAScript programs.

When a stream of Unicode characters is to be parsed as an ECMAScript program, it is first converted to a stream of input elements by repeated application of the lexical grammar; this stream of input elements is then parsed by a single application of the syntax grammar. The program is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the goal nonterminal *Program*, with no tokens left over.

Productions of the syntactic grammar are distinguished by having just one colon "`:`" as punctuation.

The syntactic grammar as presented in clauses 11, 12, 13, and 14 is actually not a complete account of which token sequences are accepted as correct ECMAScript programs. Certain additional token sequences are also accepted, namely, those that would be described by the grammar if only semicolons were added to the sequence in certain places (such as before line terminator characters). Furthermore, certain token sequences that are described by the grammar are not considered acceptable if a terminator character appears in certain "awkward" places.

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

5.1.5 Grammar Notation

Terminal symbols of the lexical and string grammars, and some of the terminal symbols of the syntactic grammar, are shown in **`fixed width`** font, both in the productions of the grammars and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a program exactly as written.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by one or more colons. (The number of colons indicates to which grammar the production belongs.) One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

WithStatement **:**

with (*Expression* **)** *Statement*

states that the nonterminal *WithStatement* represents the token **`with`**, followed by a left parenthesis token, followed by an *Expression*, followed by a right parenthesis token, followed by a *Statement*. The occurrences of *Expression* and *Statement* are themselves nonterminals. As another example, the syntactic definition:

ArgumentList **:**

AssignmentExpression
ArgumentList **`,`** *AssignmentExpression*

states that an *ArgumentList* may represent either a single *AssignmentExpression* or an *ArgumentList*, followed by a comma, followed by an *AssignmentExpression*. This definition of *ArgumentList* is *recursive*, that is, it is defined in terms of itself. The result is that an *ArgumentList* may contain any positive number of arguments, separated by commas, where each argument expression is an *AssignmentExpression*. Such recursive definitions of nonterminals are common.

The subscripted suffix "*opt*", which may appear after a terminal or nonterminal, indicates an *optional symbol*. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it. This means that:

VariableDeclaration **:**

Identifier Initializer$_{opt}$

is a convenient abbreviation for:

VariableDeclaration **:**

Identifier
Identifier Initializer

and that:

IterationStatement **:**

for **(** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

is a convenient abbreviation for:

IterationStatement **:**

for **(** **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*
for **(** *Expression* **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$ **)** *Statement*

which in turn is an abbreviation for:

IterationStatement **:**

for **(** **;** **;** *Expression*$_{opt}$ **)** *Statement*
for **(** **;** *Expression* **;** *Expression*$_{opt}$ **)** *Statement*
for **(** *Expression* **;** **;** *Expression*$_{opt}$ **)** *Statement*
for **(** *Expression* **;** *Expression* **;** *Expression*$_{opt}$ **)** *Statement*

iTeh STANDARD PREVIEW
(standards.iteh.ai)
ISO/IEC 16262:1998
https://standards.iteh.ai/catalog/standards/sist/bac8e292-7136-4211-9516-2990340e30a0/iso-iec-16262-1998

which in turn is an abbreviation for:

IterationStatement **:**

for **(** **;** **;** **)** *Statement*
for **(** **;** **;** *Expression* **)** *Statement*
for **(** **;** *Expression* **;** **)** *Statement*
for **(** **;** *Expression* **;** *Expression* **)** *Statement*
for **(** *Expression* **;** **;** **)** *Statement*
for **(** *Expression* **;** **;** *Expression* **)** *Statement*
for **(** *Expression* **;** *Expression* **;** **)** *Statement*
for **(** *Expression* **;** *Expression* **;** *Expression* **)** *Statement*

so the nonterminal *IterationStatement* actually has eight alternative right-hand sides.

If the phrase "[no *LineTerminator* here]" appears in the right-hand side of a production of the syntactic grammar, it indicates that the production is *a restricted production*: it may not be used if a *LineTerminator* occurs in the input stream at the indicated position. For example, the production:

ReturnStatement **:**

return [no *LineTerminator* here] *Expression*$_{opt}$ **;**

indicates that the production may not be used if a *LineTerminator* occurs in the program between the **return** token and the *Expression*.

Unless the presence of a *LineTerminator* is forbidden by a restricted production, any number of occurrences of *LineTerminator* may appear between any two consecutive tokens in the stream of input elements without affecting the syntactic acceptability of the program.